Indexes driven mechanism for grouped SQL queries

Radosław Boroński, Grzegorz Bocewicz

Department of Electronics and Computer Science, Koszalin University of Technology, Poland

Abstract: This paper discusses the problem of automatic minimization of a response time for a database workload by a proper choice of indexes on production systems. The main objective of our contribution is to illustrate the database queries as a group and search for good indexes for the group instead of an individual query. We present queries block relation conditions for applying the concept of grouped queries index selection. We also introduce genetic algorithm that we use in experimental test. Numerical results are presented to show quality of the recommended approach.

Keywords: index, genetic algorithm, database, grouped queries

1. Introduction

Getting database search result quickly is one of the crucial optimization problems in a database processing.

For the purposes of support and automation of production systems facilities of large corporations (General Motors), there is a continuing need for analysis of multi-source unsorted data. Companies with decision-making support tools take data feeds directly from relational databases or data warehouses. In a data warehouse to which external databases are linked and analyze terabytes of data in 24 × 7 window, there is a need to extract information from its various areas quickly and efficiently. In order to achieve this, a data mining processes are performed consisting SQL queries that in turn gather information from multi-dimensional data area. Such SQL queries block runs in production support systems periodically. Because of its repeatability and reproducibility, it is important that this process runs efficiently and automatically and has no negative impact on the overall production process.

It is a common practice to minimize the database search process at minimal cost. A database administrator (or a user) may redesign the physical hardware structure or reset the database engine parameters or try to find good table indexes for a current query. Nowadays, most vendors offer automated tools to tune the physical design of a database as part of their products to reduce the DBMS's total cost of ownership [3]. As adding more CPUs or memory may not always be possible (i.e. limited budget) and maneuvering within hundreds of database parameter may lead to a temporary solution (wrong settings for other database queries), index optimization should be considered as first.

Indexes are optional data structures built on tables. Indexes can improve data retrieval performance by providing a direct access method instead of the default full table scan retrieval method [7]. In the simple case, each query can be answered either without using any index, in a given answer time or with using one built index, reducing answer time by a gain specified for every index usable for a query [14]. Hundreds of consecutive database queries together with a large amount of data involved lead to a very complex combinatorial optimization problem. In production systems with relational databases or data warehouses, consisting hundreds of millions records of unsorted data, there is a need to improve the response time. Time needed to obtain result of non-indexed database tables may take many hours. Such a long response time is not acceptable for production systems. Indexes in such cases may reduce the response time of 50 % (depending on which columns are used for the indexing). The classic index selection method focuses on a tree data structure, which could limit the search area as much as possible. Literature discusses such B-tree types as:

- Sorted counted B-trees, with the ability to look items up either by key or by number, could be useful in database-like algorithms for query planning [4],
- Balanced B*-tree that balances more neighboring internal nodes to keep the internal nodes more densely packed [12],
- Counted B-trees with each pointer within the tree and the number of nodes in the subtree below that pointer [19].

The B-tree and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [11]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

In this paper, we discuss a simple variant of the B-tree (balanced B*-tree, proposed by Wedekind [20]), especially well-suited for use in a concurrent database system [15].

While the selection of indexes structure has a very important role in the design of database applications, one should plan the indexes structure and number of indexes at the early stage of database development operation. In such situations more important is to ask a question: "how to choose a set of indexes for the selected query sets?" It turns out that the proper selection of indexes can bring significant benefits for the database query execution time. Typical approaches found in the literature focus mainly on the search indexes only for a single column or a single query [4, 9–10, 16–17]. This paper presents

an approach associated with the search query indexes for groups called blocks.

In this case we will consider B-tree indexes. A B-tree index allows fast access to the records of a table whose attributes satisfy some equality or range conditions, and also enables sorted scans of the underlying table [18]. Also, we focus on production systems databases with the same SQL queries repeated periodically. By doing so, we eliminate database queries' low selectivity factor where no good indexes could be found due to changing queries sets.

The rest of the paper is organized as follows: in Section 2, we describe a problem statement. In Section 3, we briefly present a classic index selection approach together with simple examples that illustrate the subject. In Section 4, we demonstrate a new method of grouped queries index selection and compare examples results with classic approach. In Section 5, we present a genetic algorithm approach we use for good indexes selection. Test and comparisons with commercial tools results are presented in Section 6. Section 7 presents our conclusions and further works.

2. Problem statement

The aim of this work is to suggest an approach of multi-queried SQL block where a sub-optimal or optimal solution is to be found that gives decision makers some leeway in their decisions. The main goal is to choose a subset of given indexes to be created in a production system database, so that the response time for a given database workload together with indexes used to process queries are minimal.

The index selection problem has been discussed in the literature. Several standard approaches have been formulated for the optimal single-query and multi-query index selection. Some past studies have developed rudimentary on-line tools for index selection in relational databases, but the idea has received little attention until recently. In the past year, on-line tuning came into the spotlight and a more refined solution was proposed. Although these techniques provide interesting insights into the problem of selecting indexes on-line, they are not robust enough to be deployed in a real system [18]. The problem is known in a literature as Index Selection Problem (ISP). According to [8] it is NP-hard. Note that in practice the space limit in the ISP is soft, because databases usually grow, thus the space limit is specified in such a way that a significant amount of storage space remains free [13].

In a real life scenario, for thousands database queries compromising hundreds of tables and thousands of columns, the search space is huge and grows exponentially with the size of the input workload. Considered the case of Index Selection Problem, it can be defined in the following way: A set of tables is given:

$$T = \{T_1, \dots, T_i, \dots, T_n\}, \tag{1}$$

described by a set of columns included in the tables:

$$K = \{k_{1,1}, \dots, k_{1,l(1)}, \dots, k_{i,j}, \dots, k_{n,1}, \dots, k_{n,l(n)}\} \tag{2}$$

where:
$k_{i,j}$ is a j-th column of table T_i. Each column $k_{i,j}$ corresponds to a set of values $V(k_{i,j})$ (tuples set) included in this column.

For the set of tables T various queries Q_i can be formulated (in SQL these are SELECT queries). These queries are put against the specified set of columns $K_i^* \subseteq K$. The result of query Q_i is the following set:

$$A_i \subseteq \prod_{k_{i,j} \in K_i^*} V(k_{i,j}) \tag{3}$$

where:
$\prod_{i=1}^{n} Y_i = Y_1 \times Y_2 \times \dots \times Y_n$ is a cartesian product of sets $Y_1, \dots, Y_n$.

For a given database DB, it is taken into account that A_i is a result of the following function:

$$A_i = Q_i(K_i^*, Op(DB)) \tag{4}$$

where:
K_i^* is a subset of used columns, $Op(DB)$ is a set of operators available in database DB of which relation describing query Q_i is built.

The time associated with the determination of the set A_i depends on the DB database used (search algorithms, indexes structures) and adopted set of indexes $J \subseteq \mathcal{P}(K_i^*)$ (where $\mathcal{P}(K_i^*)$ is a power set of K_i^*). It is therefore assumed that the query execution time Q_i in given database DB, is determined by the function $t(Q_i, J, DB)$. In short, the value of execution time for query Q_i, data base DB and set of indexes J will be define as $t_i(J)$.

In the context of the so-defined parameters, a typical problem associated with the ISP responds to the following question:

What set of indexes $J \subseteq \mathcal{P}(K_i^*)$ *minimizes the query* Q_i *execution time* $t_i(J) \to min$*?*

When a multi-component set of queries $Q = \{Q_1, \dots, Q_m\}$ is considered, the question takes the following form:

What set of indexes $J \subseteq \mathcal{P}(K^*)$ *minimizes the queries block* Q *execution time* $\sum_{Q_i \in Q} t_i(J) \to min$*?*

3. Classic index selection approach

Classic index selection approach focuses on an individual query and tries to find good index or indexes set for tables in a single query in a given block. Such an approach does not take into consideration queries in a block as a whole. By doing so, a database user may expose database to create excess number of indexes that could be redundant or not used for more than one query in an examined block. This could also result in utilizing too much disk space and time needed for the indexes creation. Finding a good index group for a large database queries block was never an easy task and usually users and database administrators rely on their experience and good practice. In the

commercial use, one may find tools that support the index selection process, such as SQL Access Advisor [6], Toad and SQL Server Database Tuning Advisor [1].

Let us consider three examples where a group of three database queries $Q = \{Q_1, Q_2, Q_3\}$ is given:
Q_1: *SELECT * FROM* T_1, T_2 *WHERE* $k_{1,1} < k_{2,2}$ *AND* $k_{1,3}$ = *[const]*,
Q_2: *SELECT * FROM* T_2, T_3 *WHERE* $k_{2,2} = k_{3,2}$,
Q_3: *SELECT * FROM* T_2 *WHERE* $k_{2,1} >$ *[const]*.

Interpretation of this type of queries (according to (4)) is as follows:
Q_1: searching for a set of triples: $A_i = \{(a,b,c): a \in V(k_{1,1}), b \in V(k_{2,2}), c \in V(k_{1,3}); a < b, c = [const]\}$,
set $K_1^* = \{k_{1,1}, k_{2,2}, k_{1,3}\}$.
Q_2: searching for a set of pairs: $A_i = \{(a,b): a \in V(k_{2,2}), b \in V(k_{3,2}); a = b\}$,
set $K_2^* = \{k_{2,2}, k_{3,2}\}$.
Q_3: searching for a set: $A_i = \{a: a \in V(k_{2,1}); a = [const]\}$,
set $K_3^* = \{k_{2,1}\}$.

Tables T_1, T_2, T_3 contain $1*10^6$ records each. No indexes are built on either table: $J = \emptyset$. With the first test run, database returned following response times: $t_1(J) = 2040$ s, $t_2(J) = 3611$ s, $t_2(J) = 345$ s respectively, resulting in full table scans for each Q. Queries Q ran on database Oracle 11.2.0.1 installed on server with Red Hat Enterprise Linux 6 operating system with 64 GB memory and ASM used for disk storage.

The classic approach requires treating every database query individually. Hence, indexes are built $k_{1,1}$ and $k_{1,3}$ on table T_1; $k_{2,1}$, $k_{2,2}$ on table T_2; $k_{3,2}$ on table T_3. This kind of indexes are represented by the set $J = \{\{k_{1,1}, k_{1,3}\}, \{k_{2,2}\}, \{k_{3,2}\}, \{k_{2,1}\}\}$ containing four sets. Each element (set) of J contains the columns that are used to build the indexes. For example, the set $\{k_{1,1}, k_{1,3}\}$ means that we have to build one index for columns $k_{1,1}, k_{1,3}$.

The set of indexes J is built for three different tables, resulting in use of 2 GB of additional disk space. With the second test run, database returned following response times $t_1(J) = 2612$ s, $t_2(J) = 2580$ s, $t_3(J) = 5$ s respectively. As the response time is better by approximately 10 %, there is still unreasonable disk space used and time needed for creating 4 large indexes. Creating 4 indexes forced query optimizer to use them, and instead of decreasing Q_1 execution time, it got increased. This is because the optimizer decided to read $k_{1,1}$ column index content first and because it could not find values for $k_{1,3}$ column, it performed a full table scan for table T_1. Examples show that selected indexes may increase the query execution performance, where in other cases may have the opposite effect.

4. Grouped queries approach

In this paper, we focus on related queries group and because of this relation on the number of indexed columns. We take into account the search for a good index for the entire queries block. We propose a new approach by using multi-query SQL block selection. Such a block consists of tabular relations between queries, meaning that the number of tables columns used in a previous query is present in other queries. The proposed approach could be an alternative to the classic index selection method, where one common index set can be found. The grouped queries approach has to be studied for its effectiveness and authenticity via a series of numerical tests. Furthermore, in order to compare the performance of the method, we use commercial tools.

For previous examples, we suggest to create a pool of all columns taking part in all queries in a group and build sub-optimal indexes set for queried tables. Such a task involves creating a weighted list that will include all the index candidate query-related columns and their number of occurrence in the examined queries block:

$$KW = \left((k_{1,1}, 1), (k_{1,3}, 1), (k_{2,1}, 1), (k_{2,2}, 2), (k_{3,2}, 1)\right) \quad (5)$$

Of course, only $k_{2,2}$ column (marked by the box in (5)) is a query-related candidate column that could be used for the index creation. Nevertheless, other columns from remaining tables could also be revised. In that context, we suggest to create a composite index for the same table T_2 on columns $k_{2,1}$ and $k_{2,2}$ $J = \{\{k_{2,1}, k_{2,2}\}\}$. By doing so, the user not only speeds up block execution but also saves significant volume of disk space. With the third test run, database returned following response times $t_1(J) = 1235$ s, $t_2(J) = 2430$ s, $t_3(J) = 5$ s, respectively, decreasing total execution time of 35 % and saving disk space of 60 %. This is due to the fact that only an index is used or a full table scan for non-indexed table resulting in smaller response times for Q_1 and Q_2. The database optimizer does not need to perform an additional read operation (separate for index and if values not found and separate for a table). This proves that indexes should be selected with care.

Determining the answers to a set of queries can be improved by creating some indexes.

Classic index selection focuses on each query individually and the final indexes set is a sum of indexes sub-sets for each query.

We show that groups of queries, one can get better indexes set if such group is treated as a whole.

Grouped queries index search can only benefit and has an advantage over single query search, only if queries in the group satisfy the condition of mutual dependence. Queries Q_1, Q_2, Q_3, from previous examples are dependent so the below statement applies. Such dependency must be clearly defined.

In the present case, the dependence set of queries Q is determined by connectivity of hypergraph $G(Q)$.

Figures 1 presents an example of a hypergraph for considered queries.

In this type of graph, vertices represent the columns used in queries Q, edges connect those vertices that together create table T_a (dashed line hyper edge) or

related queries Q_i (solid line hyper edge). For example, hyper edge connecting vertices $k_{1,1}, k_{2,2}, k_{1,3}$ represents relation with query Q_1.

It is assumed that the queries set Q is related if corresponding hypergraph $G(Q)$ is consistent.

In this context, the group queries indexes set creation can benefit compared to classic index selection only for related sets.

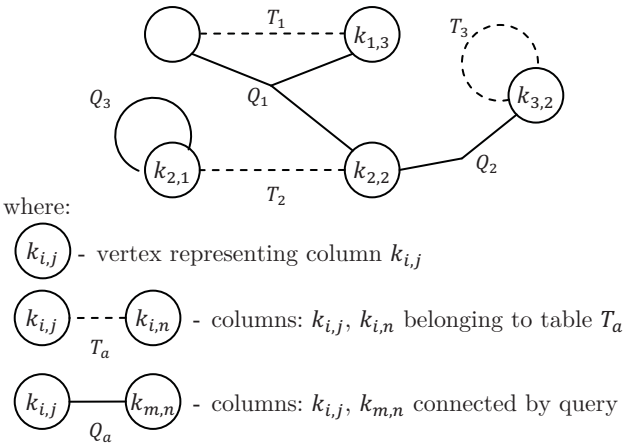

Fig. 1. Hypergraph for considered set of queries Q
Rys. 1. Hipergraf dla rozważanej grupy zapytań Q

As a counterexample, a group of three database queries $Q^* = \{Q_1^*, Q_2^*, Q_3^*\}$ is given:

Q_1^*: *SELECT * FROM* T_1, T_2 *WHERE* $k_{1,1} > k_{1,2}$,
Q_2^*: *SELECT * FROM* T_2, T_3 *WHERE* $k_{2,1} = k_{3,2}$,
Q_3^*: *SELECT * FROM* T_4 *WHERE* $k_{4,1} >$ *[const].*

Figure 2 presents the example of a hypergraph for considered queries Q^*. This kind of hypergraph is inconsistent. For this reason, queries Q^* are treated as the unrelated queries.

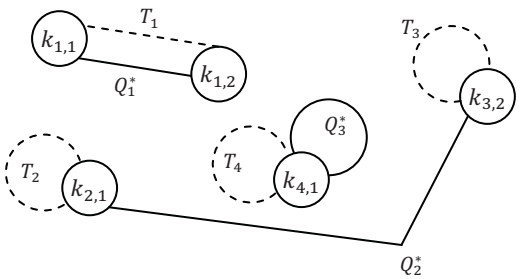

Fig. 2. Hypergraph for considered set of queries Q^*
Rys. 2. Hipergraf dla rozważanej grupy zapytań Q^*

Unrelated queries for the index selection process means they cannot be treated as a group. In such cases the best index set is a set determined for each query individually:

$$J^* = \left\{\{k_{1,1}, k_{1,2}\}, \{k_{2,1}\}, \{k_{3,2}\}, \{k_{4,1}\}\right\} \quad (6)$$

A weighted list for Q^* includes all the index candidate columns:

$$KW^* = \left((k_{1,1}, 1), (k_{1,2}, 1), (k_{2,1}, 1), (k_{3,2}, 1), (k_{4,1}, 1)\right) \quad (7)$$

One can notice there are no query-related candidate columns (single column occurrence) that could be used for the grouped queries index set creation. Each table T_i will have to be indexed separately for each individual query Q^*.

5. Genetic algorithm mechanism

There are different approaches towards index search with genetic algorithm [GA] use. Some focus on the index selection for a single query through index configuration change [14]. Other focus on change of the queries execution plan [13], which is not robust enough to be deployed in a real system, because it is based on estimation and not on a real execution time calculation. We propose an approach that is based on the analysis of multiple queries block execution (feature of production systems). Such an approach allows searching for indexes dedicated for particular production environment.

As we pursue to find a set of indexes $J \subseteq \mathcal{P}(K^*)$ that minimize queries block Q execution time $\sum_{Q_i \in Q} t_i(J) \rightarrow min$, we propose to use a genetic algorithm with a constant population size, a constant mutation rate, and no crossover. We assume that this kind of automatic, self-adaptive approach finds good block queries indexes in timely manner. Proposed algorithm consists of four steps:

5.1. Columns extraction

For each table in queries Q we eliminate columns that are not good candidates for index build (for each database engine such rules are clearly defined). Next, we create a sequences of pairs containing chosen columns and numbers of their occurrence:

$$KW = \left((k_{i_1,j_1}, h_{i_1,j_1}), (k_{i_2,j_2}, h_{i_2,j_2}), \ldots, \left(k_{i_q,j_q}, h_{i_q,j_q}\right)\right) \quad (8)$$

where:
k_{i_a,j_a} is a j_a-th column of table T_{i_a}, $a = 1 \ldots q$, and $k_{i_a,j_a} \in K^*$, $K^* = \bigcup_{i=1}^{m} K_i^*$ are a set of columns used in queries $Q = \{Q_1, \ldots, Q_m\}$, q is a number of elements KW, h_{i_a,j_a} is a number of occurrences of column k_{i_a,j_a} in queries .

The sequences (5) and (7) are examples of (8). This kind of sequence determines the initial population.

5.2. Population build

Every population of the proposed algorithm is built by a set number of individuals that are the sequences of chromosomes corresponding to tables of queries Q:

$$\Delta = (C_1, C_2, \ldots, C_r) \quad (9)$$

where:
Δ – a sequence of chromosomes,
r – number of tables corresponding to queries Q,
C_i – i-th chromosome corresponding to table T_i:

$$C_i = (g_{i,1}, g_{i,2}, \ldots, g_{i,lg\,(i)}) \tag{10}$$

where: $g_{i,j} \in \{0,1\}$ is a gene representing column $k_{i,j}$ of table T_i.

In that context, each chromosome C_i is a sequence representing columns of the table T_i used in queries Q. The value of a gene $g_{i,j}$ determines which columns belong to the set of indexes J, e.g. a value $g_{i,j} = 1$ means that column $k_{i,j}$ belongs to the set of indexes J. For example, individual Δ determining the set of indexes J (6) has the following form:

$$\Delta = \big((1,1),(1),(1),(1)\big) \tag{11}$$

where: $C_1 = (1,1)$, $C_2 = (1)$, $C_3 = (1)$, $C_4 = (1)$ are the chromosomes corresponding to tables T_1, T_2, T_3, T_4 respectively. Moreover, each individual Δ is described by time of queries Q execuiton $t(J_\Delta) = \sum_{Q_i \in Q} t_i(J_\Delta)$, where J_Δ is the set of indexes determined by Δ.

We assume that the population of individuals (9) contains the same number of elements pn in each iteration. A starting population contains one individual Δ_{KW} determined by the sequence KW and $pn - 1$ individuals as mutations of Δ_{KW}. Δ_{KW} are created as a sequence, where each chromosome C_i contains only one positive gene (a gene which corresponds to the column with the maximum occurrences of columns T_i).

5.3. Iterated evolution process

For the individuals selection we use a simple tournament selection. In each iteration, we randomly select np numbers of individuals' pairs from a given population. Among the elements of pairs we select one individual for which query block execution time best matches the fitness function ($t(J_\Delta) \to \min$). The selected sequence Δ is a new individual for the next population.

From a new population, every gene is selected and mutated (binary state change) with a probability determined by a set parameter, thus creating a new chromosome. We create a new index for table T_i for each individual's chromosome and for these indexes we measure Q queries block total execution time $t(J_\Delta)$. This closes the iteration cycle.

5.4. Stop condition check

We check when to stop the algorithm run. This can be determined by two following options:
– when a set goal is achieved: value $t(J_\Delta)$ of best individual reaches the given Ht level,
– after set run time: time of computation exceeds the given Tc value.

One must note that on number of occasions back regression may occur (no improvement in population's total execution time). In this case, we go one population back and create a new population.

The proposed genetic algorithm has been built in the Java programming environment [21] and implemented on a production and test databases for both classic and grouped queries approaches. We chose Java because of the operation system independence and vary database software connection ability (JDBC).

The next section presents the numeric results for the presented algorithm.

6. Experimental tests

In Section 4, we show two examples where the grouped queries approach may be beneficial for SQL blocks with related queries, which is blocks of queries that can be graphically represented by a consistent hypergraph (see fig. 1).

Now we use the proposed index driven mechanism for grouped queries and classic index selection approaches. In that context we carry out an experiment that involve index selection for queries block Q' represented by the consistent hypergraph (see fig. 3).

Analyzed query block Q', consists of three queries which characterize relations between columns of three database tables $T = \{T_1, T_2, T_3\}$, containing 10^7 rows each. For experimental purposes, we use Oracle database, version 10.2.0.3, installed on server with Redhat 6 operating system with 64 GB memory and ASM used for disk storage. The queries block Q' is presented (using the SQL language notation) in Table 1.

We set the algorithm's initialization parameters as follows $pn = 6$, $Ht = 247s$ and $Tc = 10^4\,s$. The experimental queries block is examined by three different tests so that good index group for each query block is found:
– index selection with use of advisory tools,
– classic index selection approach,
– grouped queries index selection approach.

In the first test we use two different index selection advisory tools. One is the Oracle SQL Access Advisor, provided together with the server database installation package. Another is TOAD package, developed by Quest company. Oracle's software has the ability to search for indexes not only for individual queries but also for a queries block (SQL Tuning Set). TOAD tool treats every SQL query within a group as an individual and indexes are selected individually, too. For two other tests (classic and grouped queries approach) we use our own index selection adaptive algorithm. The results for all three tests we carry out are shown below:

Test 1: For queries block with recommendations of index selection advisory tools the block execution time is 267 s.
Test 2: For queries block with recommendations of classic index selection approach, the block execution time is 253 s.

Test 3: For queries block with recommendations of grouped queries index selection approach, the block execution time is 245 s.

Based on the above results, differences between advisory tools, classic and grouped queries approach for blocks execution times is 22 s (8 %).

The obtained results show that for consistent hypergraph, the efficiency of the grouped queries approach against classic index selection approach also increases. It is worth noting that the commercial advisory tools seem to be not useful for related block queries. Advisors are unable to recommend any indexes whatsoever (see tab. 1). As it seems, for consistent hypergraph the effectiveness of such tools decrease.

Tab. 1. Database queries Q' and indexes recommendations
Tab. 1. Grupa zapytań Q' i rekomendacje indeksowe

Database queries set with low relations: Q_1: SELECT T3.KOL1,T3.KOL2 FROM TEST1 T1, (SELECT T2.KOL3, T2.KOL5 FROM TEST2 T2, TEST1 T1 WHERE T2.KOL3=T1.KOL5) T2, TEST3 T3 WHERE T1.KOL5 = T3.KOL4 AND T3.KOL1 = T2.KOL3 AND T3.KOL5 = ANY (SELECT T2.KOL5 FROM TEST2 T2, TEST1 T1 WHERE T2.KOL4=T1.KOL3) ORDER BY 1,2; Q_2: SELECT DISTINCT T1.KOL , T1.KOL2 , COUNT(*) FROM TEST1 T1, TEST3 T3, (SELECT T2.KOL4, T2.KOL1 FROM TEST2 T2, TEST3 T3 WHERE T2.KOL3=T3.KOL5) T2 WHERE T1.KOL1 = T2.KOL1 AND T2.KOL4 = T3.KOL4 GROUP BY T1.KOL1, T1.KOL2 ORDER BY 1 DESC; Q_3: SELECT DISTINCT T1.KOL2, T2.KOL5, COUNT(2) FROM TEST2 T2, TEST1 T1, TEST3 T3 WHERE T1.KOL4 = T3.KOL4 AND T1.KOL1 = T2.KOL3 AND T1.KOL5 > ANY (SELECT T2.KOL5 FROM TEST2 T2 WHERE T2.KOL1=1000) AND (T3.KOL3 > T2.KOL3) GROUP BY T1.KOL2, T2.KOL5 ORDER BY 1,2 DESC;	Oracle SQL Advisor + TOAD suggestion: NO INDEXES Classic index selection approach: CREATE INDEX k1_col1_col2_idx ON $T_1(k_{1,1}, k_{1,2})$; CREATE INDEX k1_col5_idx ON $T_1(k_{1,5})$; CREATE INDEX k2_col1_col3_idx ON $T_2(k_{2,1}, k_{2,3})$; CREATE INDEX k2_col3_col4_idx ON $T_2(k_{2,3}, k_{2,4})$; CREATE INDEX k2_col4_idx ON $T_2(k_{2,4})$; CREATE INDEX k3_col1_idx ON $T_3(k_{3,1})$; CREATE INDEX k3_col3_idx ON $T_3(k_{3,3})$; CREATE INDEX k3_col4_idx ON $T_3(k_{3,4})$; Grouped queries approach: CREATE INDEX k1_col1_idx ON $T_1(k_{1,1})$; CREATE INDEX k2_col1_col3_col4_idx ON $T_2(k_{2,1}, k_{2,3}, k_{2,4})$; CREATE INDEX k3_col2_col4_idx ON $T_3(k_{3,2}, k_{3,4})$;

7. Conclusions

Finding a good index or indexes set for a table is very important for every relational database that supports production processes. Not only in terms of performance but also cost aspects. Indexes can be crucial for a relational database to process queries with reasonable efficiency, but the selection of them is very difficult.

Presented examples show that there is a need for finding an automatic index selection mechanism for grouped queries rather than a classic (single query). Practice shows that grouped queries index selection gives better results and enables user to save time needed for index creation. In our example grouped queries indexes are more effective than the classic one because queries Q' satisfy the grouping condition (hypergraph $G(Q')$ is consistent (see fig. 3). One should note that the experiments we carry out are to determine indexes that minimize queries blocks execution time only. What is important in the general case are different parameters such as: index creation cost, number of indexes and disk storage allocation.

In the presented example we show a genetic algorithm use for queries block with classic and grouped queries approach. Proposed algorithm requires six populations which takes circa 8000 s to finish. Time needed to complete this task is large, however, for databases with repeating queries block run (as production systems) this is less important. Indexes selection based on grouped queries approach may be implemented during production system operation and indexes may be adjusted with next queries block run.

Our current works focus on the grouped queries index selection method with use of genetic algorithm that analyzes database queries, suggests indexes' structure and tracks indexes influence on the queries' execution time. We work on the system that will be used in an attempt to find better indexes for a critical part of long-running database queries in automatic production database environment. Buffering queries with good indexes together with their total execution time is a starting point for broader searches in future. Simple test presented in this article proves effectiveness of this method.

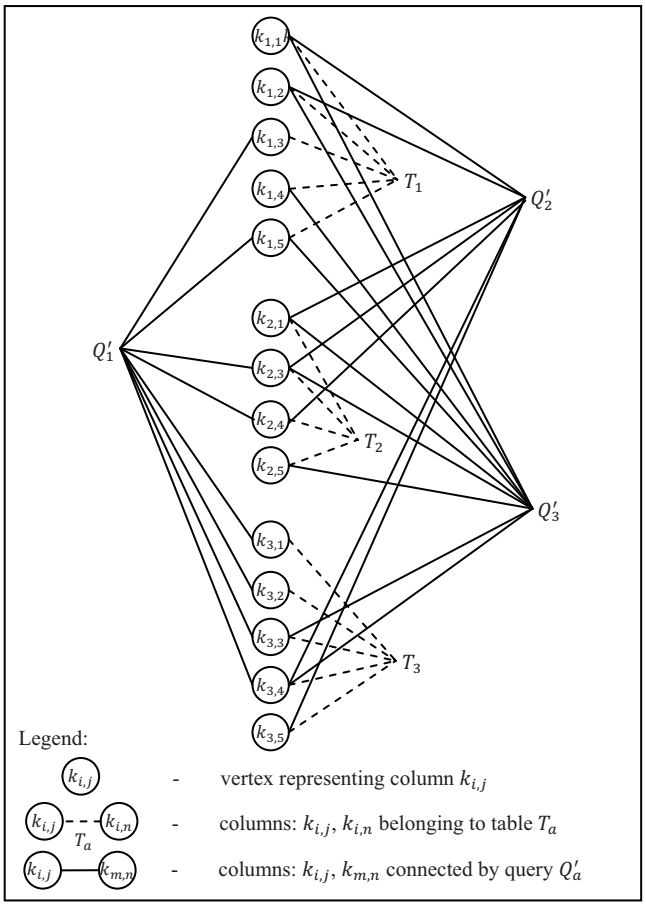

Fig. 3. Hypergraph for set of queries Q'
Rys. 3. Hipergraf grupy zapytań Q'

The developed system is scalable: there is a potentiality of combining smaller queries' blocks into larger series and finding better solution based on execution history.

References

1. Agrawal S., Chaudhuri S., Kollar L., Marathe A., Narasayya V., and Syamala M.. *Database Tuning Advisor for Microsoft SQL Server 2005*, [in:] Proceedings of the 30th International Conference on Very Large Databases, 2004.
2. Back T., *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*, Oxford University Press Oxford, UK, 1996.
3. Bruno N., Chaudhuri S., *Automatic physical database tuning: a relaxation-based approach*, [in:] Proceedings of the 2005 ACM SIGMOD international conference on Management of Data, ACM New York, NY, USA, 2005, 227–238.
4. Chaudhuri S., Narasayya V., *An efficient Cost-Driven Index Selection Tool for MS SQL Server*, Very Large Data Bases Endowment Inc, 1997.
5. Comers D., *The Ubiquitous B-Tree*, "Computing Surveys", 11 (2), DOI:10.1145/356770.356776, 123–137.
6. Dageville B., Das D., Dias K., Yagoub K., Zait M., Ziauddin M.. *Automatic SQL Tuning in Oracle 10g*, Proceedings of the 30th International Conference on Very Large Databases, 2004.
7. Dawes C., Bryla B., Johnson J., Weishan M., *OCA Oracle 10g Administration I*, Sybex, 2005, 173.
8. Finkelstein S., Schkolnick M., Tiberio P., *Physical database design for relational databases*, ACM Trans. Database Syst. 13(1), (1988), 91–128.
9. Frank M., Omiecinski M., *Adaptive and Automated Index Selection in RDBMS*, Proceedings of EDBT, 1992.
10. Gupta H., Harinarayan V., Rajaraman A., Ullman J.D., *Index Selection for OLAP*, In Proceedings of the Internatoinal Conference on Data Engineering, Birmingham, U.K., April 1997, 208–219.
11. Knuth D., *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.
12. Knuth D., Sorting and Searching, The Art of Computer Programming, Vol. 3 (Second ed.), Addison-Wesley.
13. Kołaczkowski P., Rybiński H., *Automatic Index Selection in RDBMS by Exploring Query Execution Plan Space*, Studies in Computational Intelligence, Vol. 223, Springer, 2009, 3–24
14. Kratica J., Ljubic I., Tosic D., *A Genetic Algorithm for the Index Selection Problem*, EvoWorkshops'03, Proceedings of the 2003 International Conference on Applications of Evolutionary Computing, 2003.
15. Lehman P.L., *Efficient locking for concurrent operations on B-trees*, ACM Transactions on Database Systems (TODS), Vol. 6, Iss. 4, Dec. 1981, 650–670.
16. Maggie Y., Ip L., Saxton L.V., Vijay V. Raghavan, *On the Selection of an Optimal Set of Indexes*, IEEE Transactions on Software Engineering, 9(2), March 1983, 135–143.
17. Schkolnick M., *The Optimal Selection of Indices for Files*, "Information Systems", Vol.1, 1975.
18. Schnaitter K., *On-line Index Selection for Physical Database Tuning*, ProQuest, UMI Dissertation Publishing, 2011.
19. [http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html] – Tatham S., *Counted B-Trees*.
20. Wedekind H., *On the selection of access paths in a data base system. In Data Base Management*, Klimbie J.W., Koffeman K.L., Eds. North-Holland, Amsterdam, 1974, 385–397.
21. [http://www.oracle.com/us/technologies/java/overview/index.html].

Mechanizm wyznaczania indeksów dla grupy zapytań SQL

Streszczenie: Autorzy podejmują się problemu automatycznej minimalizacji czasu odpowiedzi bazy danych na zadaną grupę zapytań SQL poprzez poprawny dobór indeksów dla systemów produkcyjnych. Głównym naszym celem jest traktowanie zapytań jako grupy i szukanie odpowiednich indeksów dla całej grupy a nie dla pojedynczego zapytania. Przedstawiamy warunki które musi spełniać grupa zapytań. Proponujemy użycie algorytmu genetycznego do poszukiwania indeksów w testach doświadczalnych. Prezentujemy wyniki testów eksperymentalnych jako uzasadnienie użycia proponowanego podejścia.

Słowa kluczowe: indeks, algorytm genetyczny, baza danych, grupa indeksów

Radosław Boroński, MSc

Radosław Boroński received his MSc degree in Computer Science and Information Technology at Technical University of Szczecin in 2002. Currently he is a doctoral student of Koszalin University of Technology, researching methods of automatic grouped queries indexes selection for relational databases and data warehouses. In professional career, he is a senior Oracle database administrator with Acxiom corporation, administrating large databases of General Motors Company.
e-mail:radoslaw.boronski@ie.tu.koszalin.pl

Grzegorz Bocewicz, PhD

Grzegorz Bocewicz obtained his MSc degree in Telecommunications from the Koszalin University of Technology, Poland, and a PhD degree in Computer Sciences from the Wrocław University of Technology, Poland in 2006 and 2007, respectively. Currently he is employed by in the Dept. of Computer Science and Management as associate professor. He is the author and co-author over 100 manuscripts including two book, international journals, and conference proceedings. His research interests are in the areas of the operational research, decision support systems, constraints programming techniques.
e-mail: bocewicz@ie.tu.koszalin.pl